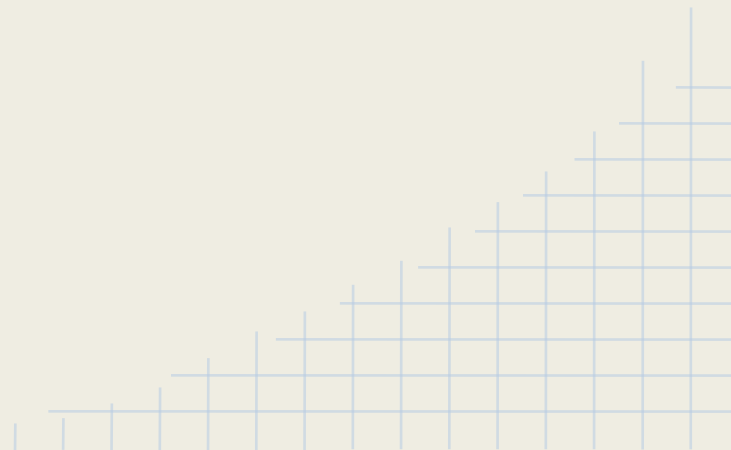
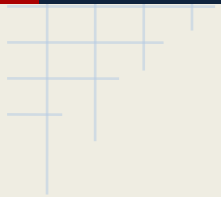


Fun with C11 generic selection expression

Zhihao Yuan <zhihao.yuan@rackspace.com>

Part I



Original use

To implement the C99 `<tgmath.h>`.

```
#define log2(a) _Generic((a),  
    long double: log2l,  
    default: log2,  
    float: log2f)(a)
```

\
\
\

The syntax

```
_Generic(controlling-expr,  
         type-name: candidate-expr,  
         default: candidate-expr,  
         type-name: candidate-expr)
```

The semantics

`_Generic(controlling-expr,
 type-name: candidate-expr,
 type-name: candidate-expr,
 default: candidate-expr)`

1. All expressions but the selected expression are in unevaluated context.
2. Each *type-name* should be a complete object type (no reference types).
3. The *controlling-expr* is decayed.
4. No two *type-name* have the same type; at most one default.
5. The result expression preserves the selected expression's value category.

The implementation

Only Clang allows generic selection expression in C++ mode.

What works:

- dependent types as candidate types

- expression SFINAE (e.g., no matched type Is Not An Error)

What doesn't:

- in the return type of a function template

Bug (or feature?):

- the controlling expression is not decayed

Implications

GCC:

```
_Generic("moew", char *: 1);
```

```
int const i = 1;
```

```
_Generic(a, int: 1);
```

Clang:

```
_Generic("moew", char[5]: 1);
```

```
_Generic(a, const int: 1);
```

```
auto& lr = i;
```

```
_Generic(lr, const int: 1);
```

Implications

GCC:

```
_Generic("moew", char *: 1);
```

```
int const i = 1;
```

```
_Generic(a, int: 1);
```

Clang:

```
_Generic("moew", char[5]: 1);
```

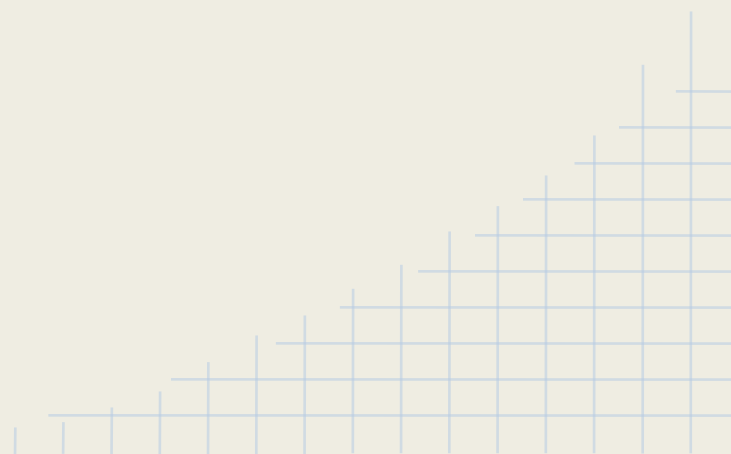
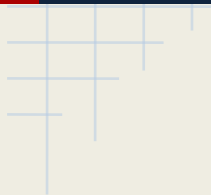
```
_Generic(a, const int: 1);
```

```
auto& lr = i;
```

```
_Generic(lr, const int: 1);
```

In the rest of the talk, Clang's semantics is used.

Part II



Examine a type

Given a type T , examine the type.

Examine a type

Given a type T, examine the type.

First try:

```
_Generic(T{}, ...)
```

Examine a type

Given a type T, examine the type.

First try:

```
_Generic(T{}, ...)
```

Second try:

```
_Generic(std::declval<T>(), ...)
```

Examine a type

Let's define a macro to simplify the use:

```
#define _Type_generic(T, ...) \
    _Generic(std::declval<T>(), __VA_ARGS__)
```

Usage:

```
_Type_generic(type, char: expr, ...)
```

Type generic literals

Given a character type `charT`, how to produce

`L"ms"` from `millisecond_suffix<wchar_t>`,
`u"ms"` from `millisecond_suffix<char16_t>`, etc.?

Type generic literals

Given a character type `charT`, produces a string literal of `charT[N]`:

```
#define _G(T, literal) _Type_generic(T,          \  
    char: literal,                               \  
    wchar_t: L ## literal,                       \  
    char16_t: u ## literal,                       \  
    char32_t: U ## literal)
```

Example:

```
_G(char16_t, "ms")    ⇒  u"ms"
```

Type generic literals

To answer the original question:

```
template <typename charT>  
charT millisecond_suffix[] = _G(charT, "ms");
```

Also works with other standard literal prefixes, suffixes, and UDLs.

Produce a type

Given a generic selection expression, produce the type.

Produce a type

Given a generic selection expression, produce the type.

```
decltype(_Type_generic(...))
```

$f :: \textit{type} \rightarrow \textit{type}$, we got a type function, anonymous.

Type function

Implement the `std::is_floating_point` trait, the traditional way:

```
template <class T> struct _is_floating_point : false_type {};  
template <> struct _is_floating_point<float> : true_type {};  
template <> struct _is_floating_point<double> : true_type {};  
template <> struct _is_floating_point<long double> : true_type {};  
template <class T> struct is_floating_point  
    : _is_floating_point<std::remove_cv_t<T>> {};
```

Type function

With generic selection expression:

```
template <typename T>
struct is_floating_point
    : decltype(_Type_generic(std::remove_cv_t<T>,
        float: std::true_type(),
        double: std::true_type(),
        long double: std::true_type(),
        default: std::false_type()))
{};
```

Control the produced type

Given some expressions, select a type to declare a variable.

First try:

```
template <typename T>
void f(T t)
{
    decltype(_Generic(..., default: t)) i;
```

Control the produced type

Given some expressions, select a type to declare a variable.

First try:

```
template <typename T>
void f(T t)
{
    decltype(_Generic(..., default: t)) i;
    // this produces T&
}
```

Control the produced type

A generic selection expression is an expression, so

```
decltype( lvalue )  ⇒ T&  
decltype( xvalue )  ⇒ T&&  
decltype( prvalue ) ⇒ T
```

Control the produced type

Don't forget that the generic selection expression preserves the value category of the selected association expression.

```
int i;  
decltype(_Generic(T, void*: i, default: 'a'))  
// int& if T is void*, otherwise char
```


Control the produced type

<code>std::remove_reference_t<decltype(_Generic(...))></code>	\Rightarrow T
<code>std::add_lvalue_reference_t<decltype(_Generic(...))></code>	\Rightarrow T&
<code>decltype(std::move(_Generic(...)))</code>	\Rightarrow T&&

Examine a value

Given a constant expression of integral, examine its value.

Examine a value

When producing values of the same type, no better than the ternary operator, nor the constexpr functions.

```
template <int i>
constexpr int fact()
{
    return _Generic(std::integral_constant<int, i>(),
        std::integral_constant<int, 0>::type: 1,
        default: fact<(i == 0 ? 0 : i - 1)>() * i);
}
```

Examine a value

But producing the heterogeneous answer is a killer app.

Enum dispatching

Given an enum definition, use it in tag dispatching.

Pros of enum:

numeric values \rightarrow computable

Pros of tags:

hierarchical relationship \rightarrow refinable

Enum dispatching

A helper to produce a complete type from an enum:

```
template <std::float_round_style i>
using enum_ct = typename std::integral_constant
    <std::float_round_style, i>::type;
```

Enum dispatching

Translate the enum to tags:

```
template <std::float_round_style i>
constexpr auto tag_of = _Generic(enum_ct<i>(),
    enum_ct<std::round_indeterminate>: round_indeterminate_tag(),
    enum_ct<std::round_toward_zero>: round_toward_zero_tag(),
    ... // warning: bad example – no refinement
```

Example:

```
f(..., tag_of<std::round_toward_zero>)
```

Examine a boolean

Given a boolean, examine the value.

```
_Generic(std::bool_constant<v>{},  
        std::true_type: ...,  
        std::false_type: ...)
```


Examine a boolean

Looks like...

```
switch (v)
{
case true: ...; break;
case false: ...; break;
}
```

Examine a boolean

Or even...

```
if (v)
```

```
...
```

```
else
```

```
...
```

Static if

Given the `if` statement and compiler optimization, why we still want

```
static if (v)
    true branch
else
    false branch
```

?

Static if

```
static if (false)
    unevaluated context
else
    potentially evaluated
context
```

```
if (false)
    potentially evaluated context
else
    potentially evaluated context
```

Static if

Only the true branch is required to be well-formed.

```
static if (false)
    r = it[i];      // as if SFINAE-out in-place
else
    r = *next(it, i);
```

Static if

Both branches are required to be well-formed, but only the true branch is (potentially) evaluated.

```
_Generic(std::bool_constant<false>{},  
        std::true_type: it[i],    // BOOM  
        std::false_type: *next(it, i))
```

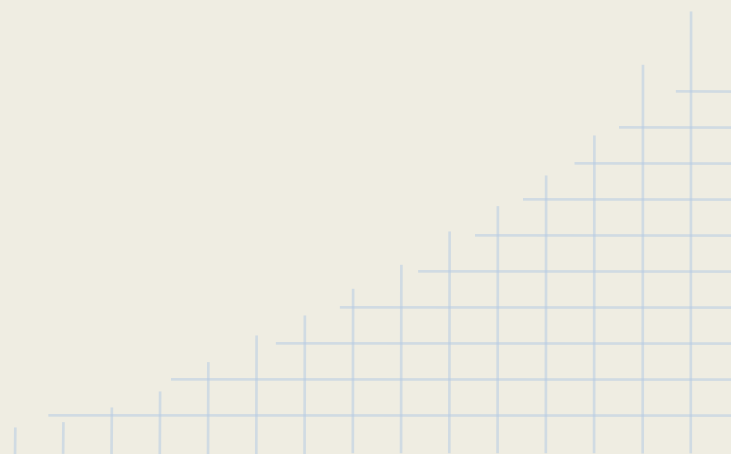
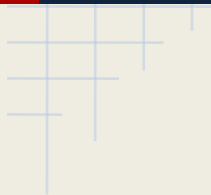
But you can generate a function object, delay the instantiation of the `operator()` — with generic lambda.

Static if

```
static_if((s), (std::is_same<T, std::string>{}),  
{  
    std::cout << s.size() << std::endl;  
}, {  
    std::cout << std::char_traits<char>::length(s) << std::endl;  
});
```

— for fun only: <https://gist.github.com/lichray/ab525cc9e970c0dfb04c>

Part III



Inspect...with

- A pattern matching syntax for C++ designed by Bjarne
- Presented at the Urbana meeting, Nov. 2014
- Powered by, probably, *Mach7*

```
inspect ( expr )  
{  
    with pattern: ...;  
    with pattern: ...;    // disclaimer: I forgot the details  
    ...  
}
```

Inspect...with

- Also claims to be able to inspect types instead of expressions (!!)

```
inspect (T)
{
    with Forward_iterator: ...;
    with Random_access_iterator: it[n];
    ...
}
```

Inspect...with

Which means, this works...

```
inspect (std::bool_constant<v>)
{
    with some-identity-concept: true-branch;
    default: false-branch;
}
```

Summary

- Generic selection expressions work like specializations inside the expressions in C++
 - useful, sometimes addictive,
 - and fun; thank you WG14.
- We want *static-if* so hard, where
 - the false branch is an unevaluated context, allowed to be ill-formed, and is discarded,
 - the true branch is a potentially evaluated context,
 - and both provide scopes.

Questions?